A Scalable Content Distribution Service for Dynamic Web Content

Seejo Sebastine
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
seejo@cs.virginia.edu

June 15, 2001

Contents

Chapter 1

Introduction

The dramatic explosion of the Internet as an information source has slowly but surely altered the notion of the Internet from that of a data communication infrastructure to a *Global Information Dissemination Service.* The World Wide Web is the dominant interface to this information service today. The continually increasing amount of content would mean escalating client access latencies, as the network infrastructure gets more and more congested. However, these latency impediments are alleviated by the key notion of web proxy caching. Web caching is currently the key performance accelerator in the web infrastructure. Client perceived performance of Internet information access depends largely on how close the information is to the client. We are aiming to provide to the client an abstraction of an infinite bandwidth Internet where everything is only a single hop away. Currently this abstraction is partially satisfied by a horde of proxy caching agents like Squid which provide this abstraction, but, only for *static* content.

However, an increasing fraction of the traffic on the web today is dynamically generated as many web sites evolve to provide sophisticated e-commerce and personalized services. Such dynamically generated content is uncacheable using present static approaches. There has been considerable interest in caching dynamic content recently [1, 2, 3, 4, 5, 6]. One possible solution to this problem is to cache, on proxies, the result web pages generated by executing the content generating scripts on the server itself. As compared to just serving static web pages, the overhead of script execution on the server is comparatively a very costly operation. This solution, therefore does not aid in reducing server load, and is not very scalable. We are therefore proposing an architecture which transparently migrates the need for computing power from the origin *content servers* to cache proxies nearest to the clients. This architecture provides the benefits of a great deal of scalability, in addition to aiding in the abstraction of the single-hop Internet, thereby improving client-observed access latencies.

In this paper, we propose a guarantee-based content-distribution service, composed of an array of trusted *"active caches"* which transparently replicate web content on demand. We do this by transparently replicating, executing and caching server scripts which generate dynamic content, on our *active caches.* Our *active caches* also cache statically generated content as regular caches do. We propose a scheme which allows content generators to subscribe to our content-distribution service and thereby benefit from the reduced latencies, lower load and improved scalability. Our service behaves as a load-balancer for the content generating web servers, since the only requests that are filtered through to them are 'preliminary' page accesses and meta-requests to maintain the data consistency. Clients also observe an improvement in access latencies since the page is served to the client from a nearer location on the Internet.

An assumption underlying our contribution is that our *active caches* are trusted by the web servers which subscribe to our service. Therefore, currently we do not incorporate any security mechanisms to authenticate the scripts that we execute locally on our *active caches*. We also assume cooperation from the subscribing servers, that allows us to make modifications to them and their content-generating scripts. We also need that the scripts conform to our API while reading and writing data. Our approach basically dynamically and transparently replicates scripts from the origin servers, hereafter called *content servers* onto our *active cache*. Since scripts are replicated on demand, subscribers to our service are only charged for scripts that are actually replicated. Any data that the script needs is also transparently moved over to the active cache. The replicated script is then executed on the *active cache*, and the results are cached. Any further requests for the results of the script (with the same input) are immediately returned from the *active cache*. Requests for the same script, but with different inputs, cause the script to be re-executed and the results of this execution to be cached for future requests, thereby reducing client-access times. Since we replicate and cache the data on which the dynamic content-generating scripts operate, our approach is most appropriate for data which does not change extremely frequently, or for applications which can tolerate a bounded amount of inconsistency in data.

The rest of this report is organized as follows. Section 2 describes the related work in this area. Section 3 describes the *"active caching"* architecture and its various components. Section 4 provides the implementation details. Section 5 concludes the report and discusses avenues for future work.

Chapter 2

Related Work

The issue of scalable on-demand *static* content replication and distribution has been addressed in numerous papers [7, 8, 9, 10, 11]. In comparison, related research is relatively scarce when dealing with *dynamic* web content. However, recently there has been considerable interest in attempting to cache dynamic content to improve the scalability and availability of web-based information services [1, 2, 3, 4, 5, 6] The methods proposed attempt to reduce the workload of dynamic content generating web servers by caching the generated content on a distributed set of reverse-proxies situated just before the web servers [2, 3, 6, 12, 13]. Specifically, Smith et al. [2] proposed a solution for cooperative caching of dynamic web content on distributed *web servers*. Clustered nodes collaborate with each other to cache and maintain result consistency using a Time-To-Live protocol. Iyengar et al. [3, 6] proposed a scheme in which servers export an API that allows individual applications to explicitly cache and invalidate application-specific content. They achieved significant performance improvement in web servers having a high proportion of dynamically generated web content. However, these approaches are limited to *web servers*. As a result, the user perceived latency improvement and backbone traffic reduction can be relatively limited as compared to proxy caching, which caches dynamic content near the clients.

Cao et al. [1] proposed an *Active Cache* which caches dynamic content at a proxy. Their solution is to attach a piece of Java code, called a *cache applet*, with each dynamic document. The cache applet is executed on the proxy whenever a request for a cached document is received. It provides the requested service without contacting the original content server. This approach is very flexible and can be used to maintain consistency in an application-specific manner while at the same time allowing dynamic updates to the existing documents. However, this flexibility is achieved with a high computational cost. It requires starting up a new Java process (VM or compiled code) for every request in order to execute the applet. Also, it is required to create and maintain a pattern-matching network for each cache-applet to keep track of result equivalence.

As regards the issue of consistency management, a lot of work has been done in the case of static web content, as is evident from the widely accepted Time-To-Live (TTL) approach. In the recent past, a considerable amount of work has also been done in managing consistency of dynamically changing information on *web servers* [2, 3, 6, 12, 13]. On a web server, strong consistency is feasible since the updates and the data dependency are available locally, which is not the case in a proxy server. Our approach addresses this issue by trading off consistency in a bounded manner for improved response time (due to proxy caching).

Chapter 3

An Active Web Caching Architecture

In this section we present our novel scalable architecture for guarantee- based on-demand replication of *dynamically changing* web content such as traffic data, stock quotes, industrial process state etc. Our architecture is designed to build upon the existing web infrastructure and does not need any components that are not already in use on the web.

Dynamic content is normally generated by *scripts* which execute on the content servers. In our approach, scripts are replicated on-demand onto a set of *"active caches"* and executed there. These *active caches* transparently replicate both static and dynamic content (also called *active content* on demand as shown in Fig. 3.1. Current web proxy caches can only replicate static web content while volatile, dynamic content is deemed uncacheable. We augment the current static caching infrastructure with our *active caches*. We designed our active caches to be compatible with current caching proxies. Only minor modifications are needed in existing proxy configuration files to interface to our novel service.

The active caches take care of the responsibility of replicating the scripts and their data on the cache, and maintaining data consistency with the content servers. Each active cache can communicate with the other caches, with the client and with the rest of the Internet. Each *active cache* is composed of two primary components; a standard static web *proxy cache* and an *active server*, as shown in Fig. 3.2. The *proxy cache* handles the job of caching static content and is already a part of the existing web infrastructure, while the *active server* is a new component that

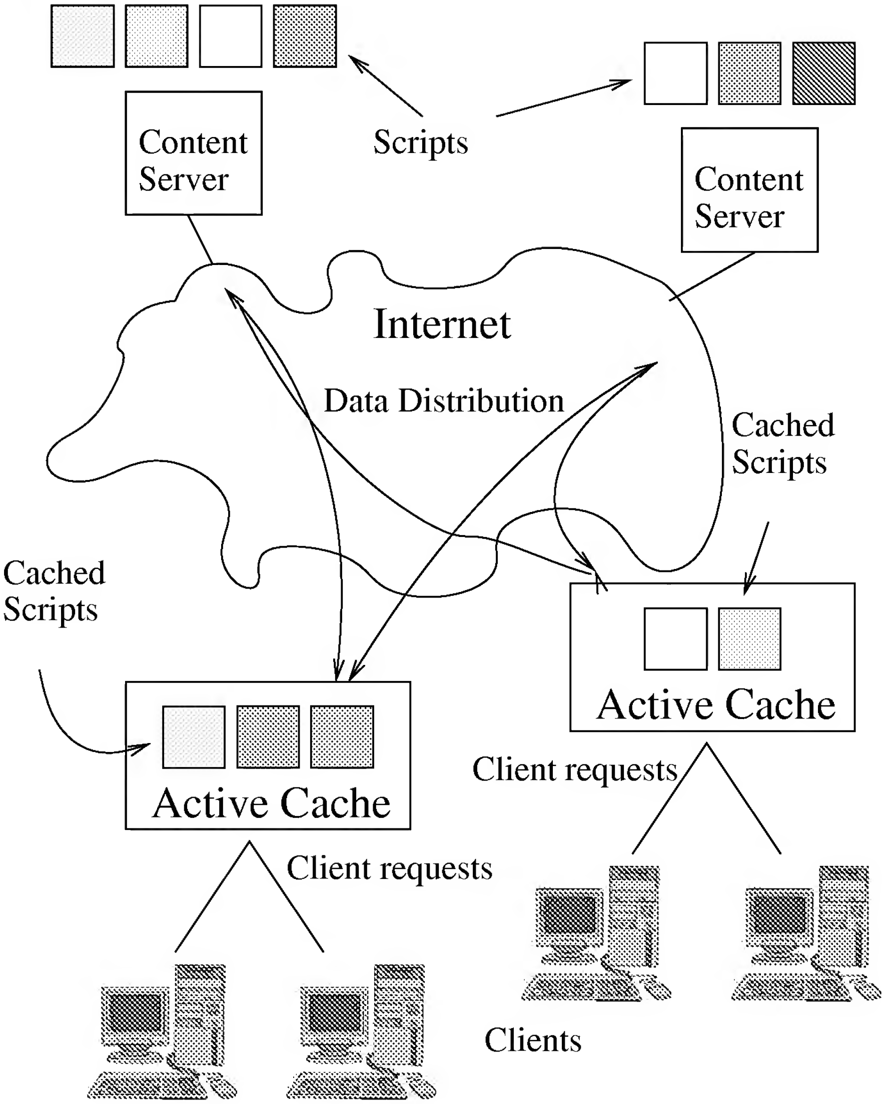

Figure 3.1: Architectural View

handles the job of executing one or more types of related dynamic content, e.g. cgi, fast-cgi, ASP, etc. Hence, each *active cache* can have multiple *active servers* to handle different types of dynamic content, such as *cgi, ASP* etc.

We have used a modified version of the freely available Squid proxy cache [14] as our *proxy cache* component and the Apache web server [15] as our *active server*. Both Squid and Apache are largely used in the web infrastructure today. Clients contact the Squid cache component with content requests. If the requests are for static content, Squid directly contacts the *content server*, retrieves the results, and caches them for future accesses. However, if the requests are for dynamic content, Squid uses its redirector component to redirect them to the Apache *active server* component, which then executes the script to generate the requested content.[1]. Since both the *proxy* and *active server* components are either running on the same physical machine or on machines on a local network, the overhead involved is far less than going over the Internet backbone to the *content server*. In addition, Squid also temporarily caches the results of the execution of the script. Hence requests in the immediate future for the same published values are served from the Squid cache (if they are still valid), thereby reducing the client-perceived latency.

The other main components of the architecture as shown in Fig. 3.2 are a *URL rewriting engine*, the *Consistency Manager module* and a *script cache*. The URL rewriting engine handles the job of fetching scripts from the content server, if they are not locally available. The script cache caches these scripts until they are no longer valid. The Consistency manager implements the consistency management algorithm. The *active server* is only visible to the *active cache* and is not accessible to the outside world. Hence it receives only those requests that have been forwarded to it by the Squid proxy cache.The *Consistency Manager* component maintains consistency by periodically checking the status of the data on the *content server* and fetching it if it has been modified.

The *content server* also requires a few modifications in order to recognize requests from active caches. Upon receiving a request for a script from an active cache, its *dynamic content module* forwards the script and its data to the active cache, instead of executing it locally. When the requested script is received, the active cache executes it locally (in the *Script Execution Module*) and also caches

[1] The script may be fetched from the *content server*, if it is not available locally

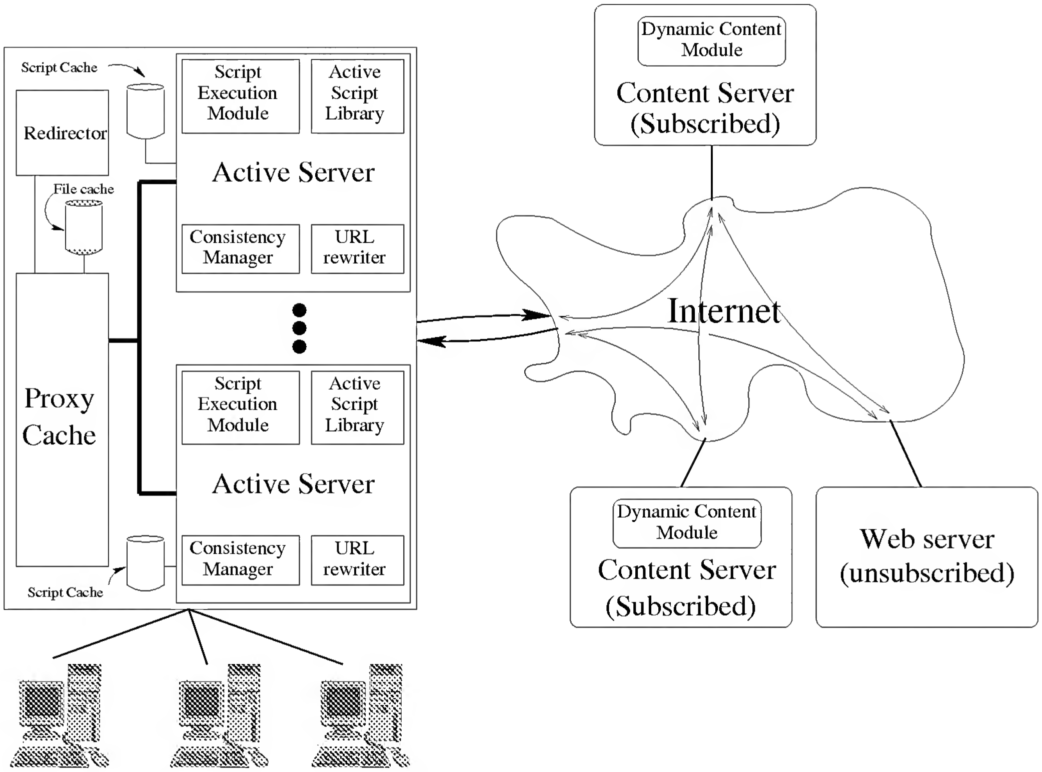

Figure 3.2: Components of an Active Cache

it in its *script cache*. In order for the scripts to interact with our system we require them to adhere to our *"Active Script API"*, implemented by the *"Active Script Library"*.

When a script is fetched for the first time it is registered with the *Consistency Manager* so that its data can be fetched periodically. Future requests for that script cause the *active server* to check with the *Consistency Manager* to see if any of the data that the script depends on has been modified. In that case the *Script Execution Module* re-executes the script and returns the new results, else it just returns the cached results.

Chapter 4

Implementation

In this section we discuss the implementation details of our architecture and protocol. The system consists of two main components: the *proxy cache* and the *active server* as seen in Fig. 3.2. It also requires some modifications to the *content server*. We chose to use the freely available Squid proxy cache (version 2.3) [14] as the *proxy cache* component and the Apache web server (version 1.3.9) [15] as the *active server* component. Sections 4.1, 4.2 and 4.3 describe the implementation details of the *proxy cache, active server* and *content server* respectively. Finally Section 4.4 describes the *Active Script API* in detail.

4.1 Squid Proxy Cache

The Squid proxy cache performs the task of channeling all requests for active content to the *active server*. It does this by selectively rewriting the client requested URLs to point to the *active server* instead of the original *content server*. This task is performed by its *redirector* component. Requests for static content are not redirected and are directly forwarded to the requested *content server*. For example, if the original request was to:

```
http://www.xyz.com/cgi-bin/hello.cgi
```

and the *active server* was at:

```
http://as1.cs.virginia.edu:8080/
```

then the request would be redirected to:

```
http://as1.cs.virginia.edu:8080/cgi-bin/hello.cgi
```

4.1.1 Redirector module

We use the redirection capabilities available in Squid for the redirector component. Squid has a very useful feature that allows a third-party redirector to be used. Hence, we used a tool available under the GNU General Public License; *JESred* version 1.2 [16], as our redirector. We chose *JESred* for this purpose, over other more popular redirectors such as *Squirm* [17], since it is about 2-3 times faster, and is highly configurable. Squid is configured to use *JESred* as its redirector by adding the following lines to Squid's configuration file:

```
redirect_program <path>/jesred
redirect_children 10
redirect_rewrites_host_header off
```

In the list above, the second option dictates how many external redirector processes can be run in parallel to handle the redirection requests. If too few redirectors are started, Squid will have to wait for them to process a backlog of URLs, thereby slowing it down. In contrast, if too many of them are started, they use up RAM and other system resources. By default, Squid rewrites any HTTP "Host:" headers in redirected requests to point to the new host to which the URL has been redirected, i.e. for example if an HTTP request:

```
GET http://www.xyz.com/cgi-bin/hello.cgi HTTP1.1
Host: www.xyz.com
```

gets redirected to *http://as1.cs.virginia.edu:8080/cgi-bin/hello.cgi* then *by default* the HTTP request would change to :

```
GET http://as1.cs.virginia.edu:8080/cgi-bin/
    hello.cgi HTTP1.1
Host: as1.cs.virginia.edu:8080
```

However, we turn this option off, (line 3 above) because we would like to remember the name of the host for whom the request was originally meant, to eventually forward the request to that host. Therefore the HTTP request in our configuration would be:

```
GET http://as1.cs.virginia.edu:8080/cgi-bin/
    hello.cgi HTTP1.1
Host: www.xyz.com
```

The configuration of *JESred* has two parts; configuring the Access Control List and configuring the Redirector Rules. The Access Control List controls the IP addresses from which *JESred* will accept requests. It is configured to be the same as those that Squid accepts. The redirection rules specify how the accepted URLs are to be redirected. As part of *JESred*'s Redirection rules, it is configured to ignore URL's that end in .html, .htm, .shtml, .java, .jpg etc. Hence all requests for static content are not redirected by *JESred,* and are handled by Squid just like it handles any static content. In contrast, URLs which contain strings like *"cgi-bin", ".asp" etc.* are redirected by *JESred* to the *active server.* This is done by adding a rule, similar to the one given here:

```
regex ^http://[^\]*/cgi-bin/(.*$) http://as1.cs.
      virginia.edu/cgi-bin/\1 /cgi-bin/
```

This rule essentially rewrites URLs that contain the string *"cgi-bin"*in them to be sent to *"as1.cs.virginia.edu"*, which is the URL for the *active server.* More extensive details on the syntax can be found in the *jesred* documentation [16] and in Appendix A.

4.1.2 Squid file cache

When the Squid *proxy cache* receives a request for dynamic content, it first checks to see if it already has a cached version of that content. After performing checks to determine if the cached content is still valid, it either returns the cached page, or redirects the request to the appropriate *active server* to obtain the most up-to-date version of that content and caches it in its file cache.

4.2 Active Server

When an *active server* receives a redirected request, it first checks to see if the script that has been requested is already cached locally in its *script cache.* If so, it executes the script with the current inputs and returns the results to the Squid *proxy cache.* In case, a cached version of the script is not available locally, the *active server* initiates a URL rewrite operation which is handled by the *URL rewriter* module.

4.2.1 URL rewriter module

The URL rewriter module handles the job of rewriting the (redirected) URLs that the *active server* receives to fetch the script from the original content server. The address of the content server is specified in the HTTP "Host:" header as elucidated in section 4.1.1. We have used the functionality provided by Apaches *"mod_rewrite"* module to perform the URL rewriting. A sample rewrite-rule that can be used is given below:

```
RewriteRule ^/cgi-bin/(.*$)
            http://%{HTTP_HOST}/cgi-bin/$1 [P]
```

This rule states that all URLs which are meant to be served from the *cgi-bin* directory are to be rewritten to be served from the same directory, but on the host specified by the HTTP_HOST environment variable. Since we wish to cache the script that is returned, we force this request to be a proxy request (indicated by [P]). This proxy request is handled by Apache's *"mod_proxy"* module. For example, if a request meant for:

```
http://www.xyz.com/cgi-bin/hello.cgi
```

was redirected to the *active server* at:

```
http://as1.cs.virginia.edu:8080/
```

then the *active server* on determining that the URL was for a script *hello.cgi* which is not cached locally, rewrites the URL again to:

```
http://www.xyz.com/cgi-bin/hello.cgi
```

and caches the returned results. More extensive details on the syntax can be found in Appendix B.

4.2.2 Script Execution Module

After a script has been retrieved from the *content server* and cached locally in the *script cache* it is executed by the *active server*. The script could potentially need some data for execution and this data has to be fetched from the *content server*. Currently, in order to determine which files are to be fetched, we require that all the scripts conform to the *Active Script API* stated in section 4.4. When the script is actually executed, the *Active Script Library* (Section 4.2.4) is invoked and it registers the URLs of the script and all its constituent data files with the *Consistency Manager* (Section 4.2.5). The *Consistency Manager* uses these URLs to periodically fetch the data files from the *content server*. Once a script has been cached in the *Script cache*, it is not re-executed until the data that it uses actually changes. It determines this by querying the *Consistency manager* on every client request.

4.2.3 Script Cache

When an *active server* receives a reply from a *content server,* it caches the reply (i.e the script) in its local *script cache*. We have used Apache's proxy caching capabilities (provided by the *mod_proxy* module) for this purpose.

4.2.4 Active Script Library

The Active Script Library implements the Active Script API stated in section 4.4. Each script must be linked with this library in order to function correctly. The library interacts with the *Consistency Manager Module* as mentioned in Section 4.2.2.

4.2.5 Consistency Manager

The *Consistency Manager* handles the job of maintaining the data consistent with that on the *content server*. As stated in section 4.4.1, each data file has consistency bounds specified by the *content server*. The *Consistency Manager* tries to maintain the data consistent within those time bounds.

The Consistency Manager receives two kinds of messages:

- Notification messages from the Active Script Library as each script is executing notifying it of (Script URL, Data URL) pairs. Each such pair indicates that the data file at URL "Data URL" is used by a script with URL "Script URL".

- Queries from the *Script Execution Module* asking it whether any of the data files for a particular "Script URL" have changed, thereby requiring that the script be re-executed.

On every notification message, the Script URL and Data URL are added to a script cache and data cache respectively, if they do not already exist. Each entry in the script cache maintains pointers to all the data files it uses. Similarly each entry in the data cache maintains pointers to each script that uses it. This is needed to determine which scripts need to be re-executed when a data file changes.

The Consistency Manager is implemented using a timer thread which times out at constant intervals of time (say 30 seconds). Each data file that is added to the system is added into this timer queue at a position determined by the current consistency that it requires. So for example if the current consistency being provided to a data file is 60 seconds and the current timeout index (i.e. the number of timeouts that have already occurred since the system started) is 10, then the data file will be added at index 12 (if the timeout period is 30 seconds).

At each timeout, first the system is checked for CPU overload, by reading the current load value in the system file "/proc/loadavg". If this load has exceeded a threshold, then it would not be prudent to continue fetching files at the same rate, since that would only further overload the system. Hence we use various heuristics to determine which of the files can be degraded, i.e. which of the files can be pushed to a future timeout. The heuristics used are:

1. Data files of *scripts* with least access frequency
2. Data files of *scripts* with most access frequency
3. Data files of *scripts* with best current QoS (The QoS of a script is nothing but the least value among the QoS' of its constituent data files)
4. Data files of random *scripts*

Once the system has been checked for potential overload, all the data files that need to be fetched at that timeout are queued up and fetched in parallel by different threads. As each data file is fetched, its backbone network delay is measured. If this delay has increased as compared to the delay measured when it was fetched the last time, then it means that the backbone is getting overloaded. Therefore, it would be difficult to maintain the current QoS and hence the QoS of such data files is worsened based on a heuristic. If this results in the QoS of a particular data file exceeding its maximum allowed, then it is temporarily dropped from the fetch queue. This is done by scheduling it to be fetched at a period of twice its maximum QoS. In all other cases it is added back to the timer queue to be fetched based on the current QoS that is being provided with. One fact to note is that the QoS of a data file is degraded *ONLY* if it has actually been modified on the server. This is determined by sending IF_MODIFIED_SINCE HTTP requests everytime a data file is fetched. If the file has actually been modified on the *content server* then the server would reply with the updated version of the file and its LAST_MODIFIED_DATE, or else the server would just reply with an HTTP status code of 304 (NOT_MODIFIED). In addition before a data file is actually fetched, its access frequency is compared with its current fetch period. If it is being accessed less frequently than it is being fetched from the content server, it is dropped from the cache (in the same way as specified above - by scheduling it to be fetched with a period of twice its maximum QoS).

Since all the data files are scheduled to be fetched in parallel from the content server by different threads, the outgoing link bandwidth could potentially get overloaded on account of such a multitude of requests. This would in turn lead to greater delays being observed for each data file fetched. In order to prevent this, we monitor the current bandwidth available every time a new thread is spawned to handle a data file fetch. If the available bandwidth falls below a threshold, then we *do not* actually fetch that data file at that timeout, but degrade it to be fetched at a later timeout. In a similar way we degrade a few more files until the load is reduced. We use a few heuristics to determine which files to degrade. They are:

1. *Data files* with least access frequency
2. *Data files* with maximum access frequency
3. *Data files* with best QoS
4. Random *data files*

In this way we try to reduce the number of outgoing requests at a particular timeout, therefore reducing the load on the outgoing network.

4.3 Content Server

The *content server* also requires certain modifications to its "dynamic content module" (Fig. 3.2) to support the active caching scheme. When a *content server* receives a proxy request for dynamic content, it has to be able to distinguish that the request originated from an *active cache.* In order to facilitate this, the *active server* modifies the HTTP request header before forwarding the request. It modifies the HTTP [18] Accept header, by adding the string:

```
x-script/x-activeScripts
```

to the header. When the *dynamic content module* recognizes this special extension, it comprehends that the script is not to be executed locally, but is to be returned to the requester as is. One very important fact to note is that there is no security inbuilt into this mechanism. It is assumed that the *active cache* and server have already authenticated each other using some reliable authentication mechanism. Hence a rogue cache can gain access to the script only if it can successfully authenticate itself with the server beforehand.

4.4 Active Script API

A script could be executed on the *active server* or the *content server* and hence we need to be able to differentiate between the two servers. This is done by requiring the script to specify the name of the *content server* for which it was originally written. This is done by the API:

```
ASContentServer (Content server name, port);
```

On account of this function call we are able to determine if the script is currently executing on the *content server* or the *active server* by comparing IP addresses. If the script is executing on the *content server* all file requests are local, but if the script is executing on the *active server* all file requests need to be converted to HTTP requests over the Internet. This function must be called before any of the file read/write functions can be called, or else those functions will fail.

The remaining API calls deal with file open, read, write and close:

4.4.1 File open

```
FD = ASopen(<filename>, attributes, consistency,
            minimum QoS, maximum QoS);
```

The ASopen() call is used to open an existing file, and in some cases to create it first. The “filename” parameter is the name and path of the file to be opened. Since the path specified as a parameter here actually refers to the path on the *content server*, the *Active Script Library* (Section 4.2.4) queries the *URL rewriter module* (Section 4.2.1) for the local cache path of the specified “filename”. The “attributes” indicate in what modes it can be opened, e.g: “r”, “w”, “r+” etc. The “consistency” parameter can have one of two different values as specified in Table 4.1.

Table 4.1: Consistency parameter values

Value	Meaning
“EXPIRES”	The file requires a consistency between minimum QoS and maximum QoS, i.e. the copy on the *active server* should not be older than “max. QoS” minutes and need not be newer than “min. QoS”.
“TEMPFL”	The file is a temporary file, i.e. it does not have any consistency requirements. It is created locally (if it does not exist) and is removed when the file is closed. It can be used to store temporary data values which need not be available on the next execution of the script.

The fourth and fifth parameters are ignored if the consistency is TEMPFL. This call returns a “file descriptor” that is used for further read/write calls.

4.4.2 File read

```
Status = ASread (FD, Buffer, Count);
```

This call attempts to read ’Count’ number of bytes from the file descriptor FD, into the buffer ’Buffer’. It returns the number of bytes successfully read.

4.4.3 File write

We only support read-only scripts and this “write” call is only to write temporary files specified by the consistency type “TEMPFL” in Table 4.1.

```
    Status = ASwrite (FD, Buffer, Count);
```

Similar to the ASread () call, this call attempts to write ’Count’ number of bytes from the buffer ’Buffer’ to the file ’FD’. It returns the number of bytes successfully written.

4.4.4 File close

```
    Status = ASclose(FD);
```

The ASclose() call closes the file descriptor ’FD’. In case the file is a temporary data file, it is also deleted.

Chapter 5

Future Work

Currently, we only support read-only scripts. Scripts are not allowed to modify any state on the content server. We are investigating methods to allow write access in a consistent manner. Another issue that remains to be addressed is the possibility of platform inconsistencies between the content server and the active cache. This can be solved by adding the required hardware to eliminate the platform inconsistencies. Security/authentication are also important research issues to be addressed in the future.

Acknowledgments

The work reported in this project was supported in part by the National Science Foundation under grants CCR-0093144 and CCR-0098269.

Appendix A

Squid Redirection Module (JESred)

Jesred is a very fast and highly configurable redirector for the Squid Internet Object Cache. It was derived from Chris Foote's and Wayne Piekarski's Squirm 1.0 betaB and some code from Squid itself, but is about 2-3 times faster than the original version and has some additional features.

A.1 Features

Jesred has the following features:

- It is faster than any other known Squid redirector
- It uses only a very small amount of memory
- It is able to rewrite GET and optionally ICP_QUERY requests as well
- It has one *global configuration file* (Section A.2)
- It has one *IP access control file* (Section A.3) which supports CIDR notation only (i.e. subnet/mask). Thus only URL requests from the specified subnets/clients are rewritten, if necessary.
- It has one *rewrite rules configuration file* (with regular expression matching and replacement) (Section A.4)
- It supports optional logging of common and error messages to a file

- It supports optional logging of URL rewrites to a separate log file, including the number of the rule, which has been used to rewrite the URL.
- It is able to re-read all its configuration files on the fly by sending a HUP signal to the process. Therefore one does not need to restart Squid, if one wants to:
  1. change the used log file name[s] (useful for log file rotation).
  2. enable/disable logging.
  3. enable/disable debug mode if compiled with DEBUG option.
  4. enable/disable URL rewriting of ICP_QUERY (sibling) requests.
  5. change the IP access patterns.
  6. change the redirect rules.
- JESred runs in Echo mode if there are configuration file error[s] (i.e. it always echos back a newline without any URL replacement) and therefore keeps Squid working.

A.2 Global Configuration File

The following options are supported:

- Path to Access Control Configuration File

  ```
  allow = /local/squid/etc/redirect.acl
  ```

- Path to Rewrite Rules Configuration File

  ```
  rules = /local/squid/etc/redirect.rules
  ```

- Path to log file for general, error and debug messages (empty value or commenting this out disables logging)

  ```
  redirect_log = /local/squid/logs/redirect.log
  ```

- Path to log file for URL rewrites (empty value or commenting this out disables logging of URL rewrites)

```
rewrite_log = /local/squid/logs/rewrite.log
```

- Enable/Disable debugging to redirect_log

```
debug = true/false
```

- Allows ICP_QUERY requests to be rewritten, if a rule applies.

```
siblings = true/false
```

A.3 Access Control Configuration File

The Access Control Configuration file determines the IP addresses from which requests are allowed to be rewritten. It supports only the following CIDR notation:

```
a.b.c.d/m with a, b, c, d between {0..255}
           and m between {0..32}
```

called as the IP access pattern. The IP access pattern can be prefixed with a '!' which means that the URL passed from that IP address should not be re-written.

JESred uses a linear list of IP access patterns and hence the order of the IP access patterns is important. As long as no match is found, it compares the clients IP address with the next pattern. If the end of the list is reached and no match was found, JESred echos back a new line, which indicates no URL replacement to Squid. If a client IP address matches, rewrite rules are applied immediately (i.e. no further rules are processed). For example:

```
# Do not rewrite URLs from these addresses
!141.44.251.15/32
!149.203.102.1/32

# Rewrite all URLs from:
141.44.0.0/16
149.203.0.0/16
193.175.28.0/24
```

A.4 Rewrite Rules Configuration File

The Rewrite Rules Configuration File specifies a regular expression based pattern, which if matched results in the rest of the rule being executed. Just as in the Access Control List, the rules are matched in a linear order and hence the order of the rules is important. The following configuration options can be present in this file:

- *Abort string*: If JESred encounters one of these strings at the end of the passed URL, it immediately returns and echo’s back a newline (i.e. no rewrite). These rules are referred to as ABORT rules. For example:

  ```
  abort .html
  abort .jpg
  abort .html
  abort .shtml
  abort .java
  abort .jar
  abort .htm
  ```

- *Rewrite rule regular expressions:* These regular expressions specify the pattern to be matched and the action to be performed if a match occurs. The syntax is:

  ```
  regex RE [RURL [ACCEL]]
          OR
  regexi RE [RURL [ACCEL]]
  ```

  regex indicates that the following regular expression is case-sensitive and *regexi* indicates that the following regular expression is case-insensitive. *RE* is the regular expression which has to match the passed URL to get rewritten with the following *RURL.*

  ACCEL is a string which starts with a ’∧’. If the string after the ’∧’ is NOT completely the same as the string that the passed URL starts with then the the rest of the rule is ignored. This is used as an accelerator to improve the matching speed.

If RURL and ACCEL are omitted, all URLs which match RE, are NOT rewritten. Some examples are given below:

```
regex ˆhttp://199.78.52.10/˜web_ani/.*\.gif
       http://141.44.30.2/images/dot.gif
      ˆhttp://199.78.52.10/˜web_ani/
```

This rule replaces all URLs starting with *http://199.78.52.10/∼web_ani* (note the accelerator) and ending in *.gif* with the exact URL *http://141.44.30.2/images/dot.gif*. The pattern *[.*]* is used to match any characters. If there is more than one such *[.*]* pattern in the regular expression, then each such pattern is numbered from left to right and can be used in the RURL to replace the exact same string that was matched. For example:

```
regex ˆhttp://(.*)/ads/mini/(.*)
       http://\1/adverts/large/\2
```

This rule replaces all URLs that have the string */ads/mini/* in them with the string */adverts/large/*. It does this by copying all the characters matched by the first .* into the result URL where the string \*1* exists and all characters matched by the second .* into the result URL where the string \2 exists.

Here are some more examples:

```
regexi ˆhttp://ad.doubleclick.net/ad/.*
         http://141.44.30.2/images/dot.gif
        ˆhttp://ad.doubleclick.net/ad/

regex ˆhttp://ad.preferences.com/image.*
       http://141.44.30.2/images/dot.gif
      ˆhttp://ad.preferences.com/image

regex ˆhttp://ads[0-9][0-9].focalink.com/
              SmartBanner/nph-graphic.*
       http://141.44.30.2/images/dot.gif

regex ^http://adserver.developer.com/cgi-bin/
              accipiter/adserver.exe.*
       http://141.44.30.2/images/dot.gif
       http://adserver.developer.com/cgi-bin/
              accipiter/adserver.exe

regex ^http://tracker.clicktrade.com/Tracker.*
       http://141.44.30.2/images/dot.gif
       http://tracker.clicktrade.com/Tracker

regex ^http://adforce.imgis.com/?adserv.*
       http://141.44.30.2/images/dot.gif
      ^http://adforce.imgis.com/?adserv

regex ^http://195.90.252.40/banner.*
       http://141.44.30.2/images/dot.gif
      ^http://195.90.252.40/banner

regex ^http://www.artuframe.com/partners/affiliates/
              banners.*
       http://141.44.30.2/images/dot.gif
      ^http://www.artuframe.com/partners/affiliates/
              banners
```

Appendix B

Apache Redirection Module (mod_rewrite)

The mod_rewrite module of Apache is very useful tool for URL manipulation. It uses a rule-based rewriting engine (based on a regular-expression parser) to rewrite requested URLs on the fly. It supports an unlimited number of rules and an unlimited number of attached rule conditions for each rule to provide a really flexible and powerful URL manipulation mechanism. The URL manipulations can depend on various tests, for instance server variables, environment variables, HTTP headers, time stamps and even external database lookups in various formats can be used to achieve a really granular URL matching.

It operates on the full URLs (including the path-info part) both in per-server context (httpd.conf) and per-directory context (.htaccess) and can even generate query-string parts on result. The rewritten result can lead to internal subprocessing, external request redirection or even to an internal proxy throughput.

B.1 How does mod_rewrite work?

The internal working of this module is very complex and needs to be understood clearly to avoid common mistakes.

B.1.1 API Phases

When Apache processes a HTTP request it does this in phases. A hook for each of these phases is provided by the Apache API. Mod_rewrite uses two of these hooks: the URL-to-filename translation hook which is used after the HTTP request has been read but before any authorization starts and the Fixup hook which is triggered after the authorization phases and after the per-directory config files (.htaccess) have been read, but before the content handler is activated.

So, after a request comes in and Apache has determined the corresponding server (or virtual server) the rewriting engine starts the processing of all the mod_rewrite directives from the per-server configuration in the URL-to-filename phase. A few steps later when the final data directories are found, the per-directory configuration directives of mod_rewrite are triggered in the Fixup phase. In both situations mod_rewrite rewrites URLs either to new URLs or to filenames, although there is no obvious distinction between them.

More details on the API phases can be found at:

```
http:\\httpd.apache.org/docs/mod/mod_rewrite.html
```

B.1.2 Ruleset Processing

The order of rules in the ruleset is important because the rewriting engine processes them in a special (and not very obvious) order. The rule is this: The rewriting engine loops through the ruleset rule by rule (*RewriteRule* directives) and when a particular rule matches it optionally loops through existing corresponding conditions (*RewriteCond* directives). For historical reasons the conditions are given first, and so the control flow is a little bit long-winded. See Figure B.1 for more details.

As is obvious, first the URL is matched against the *Pattern* of each rule. When it fails mod_rewrite immediately stops processing this rule and continues with the next rule. If the *Pattern* matches, mod_rewrite looks for corresponding rule conditions. If none are present, it just substitutes the URL with a new value which is constructed from the string *Substitution* and goes on with its rule-looping. But if conditions exist, it starts an inner loop for processing them in the order that they are listed. For conditions the logic is different: we don't match a pattern against

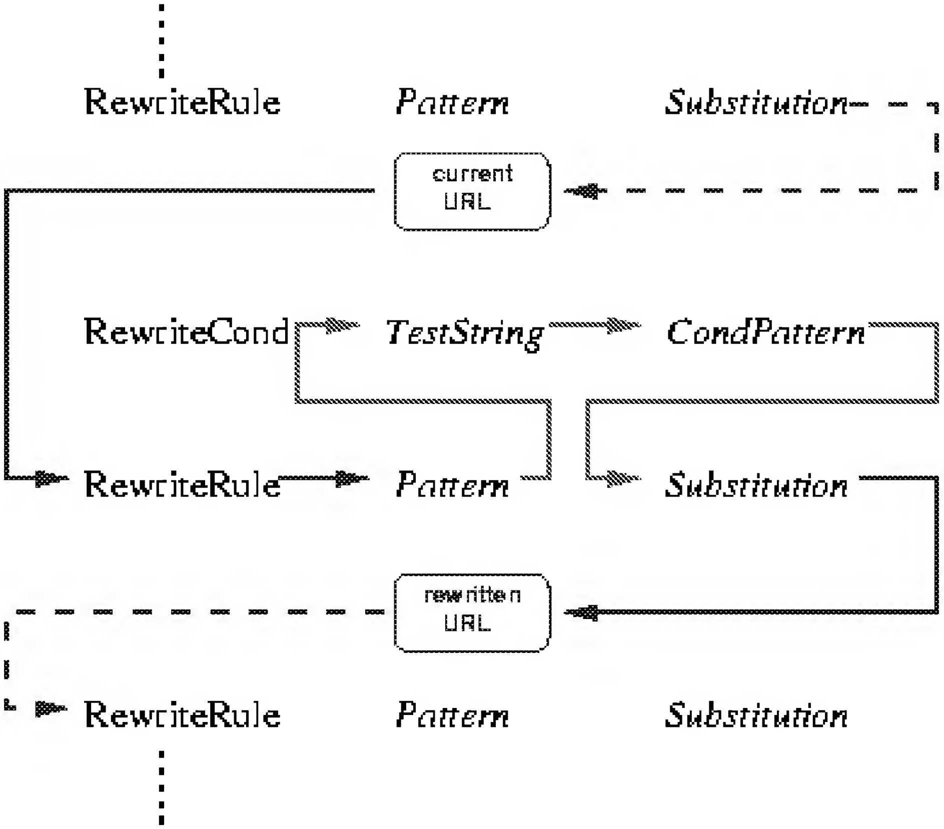

Figure B.1: The control flow through the rewriting ruleset

the current URL. Instead we first create a string *TestString* by expanding variables, back-references, map lookups, etc. and then we try to match *CondPattern* against it. If the pattern doesn't match, the complete set of conditions and the corresponding rule fails. If the pattern matches, then the next condition is processed until no more conditions are available. If all conditions match, processing is continued with the substitution of the URL with *Substitution*.

Quoting Special Characters

As of Apache 1.3.20, special characters in *TestString* and *Substitution* strings can be escaped (that is, treated as normal characters without their usual special meaning) by prefixing them with a slash ('\') character. In other words, you can include an actual dollar-sign character in a *Substitution* string by using '\$'; this keeps mod_rewrite from trying to treat it as a backreference.

Regex Back-Reference Availability

One important thing here to be remembered is that whenever you use parentheses in *Pattern* or in one of the *CondPattern*, back-references are internally created which can be used with the strings $N and %N. These are available for creating the strings *Substitution* and *TestString*. Figure B.2 shows to which locations the back-references are transfered for expansion.

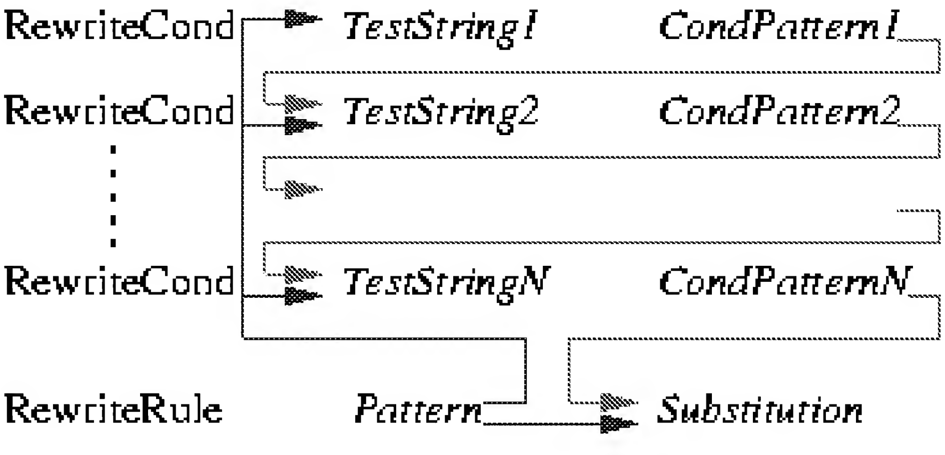

Figure B.2: The back-reference flow through a rule

B.2 Configuration Directives

Mod_rewrite provides the following configuration options:

- RewriteEngine
- RewriteOptions
- RewriteLog
- RewriteLogLevel

- RewriteLock
- RewriteMap
- RewriteBase
- RewriteCond
- RewriteRule

Extensive details on each of these options can be found at:

```
http://httpd.apache.org/docs/mod/mod_rewrite.html
```

B.3 Examples

A lot of examples with detailed explanation can be found at:

```
http://httpd.apache.org/docs/misc/rewriteguide.html
```

Bibliography

[1] P. Cao, J. Zhang, and K. Beach, "Active cache: Caching dynamic contents on the web," in *Proc. of Middleware '98*, 1998, pp. 373–388.

[2] V. Holmedahl, B. Smith, and T. Yang, "Co-operative caching of dynamic content on a distributed web server," in *Proc. oof Seventh IEEE International Symp. on High Performance Distributed Computing*, 1998, pp. 243–250.

[3] A. Iyengar and J. Challenger, "Improving web server performance by caching dynamic data," in *Proc. of USENIX Symp. on Internet Technologies and Systems*, Dec 1997, pp. 49–60.

[4] F. Douglis, A. Haro, and M. Rabinovich, "Hpp: Html macro-preprocessing to support dynamic document caching," in *Proc. of USENIX Symp. on Internet Technologies and Systems*, Dec 1997, pp. 83–94.

[5] B. Smith, A. Acharya, T. Yang, and H. Zhu, "Exploiting result equivalence in caching dynamic web content," in *Proc. of Second USENIX Symp. on Internet Technologies and Systems*, Oct 1999, pp. 209–220.

[6] J. Challenger, P. Dantzig, and A. Iyengar, "A scalable system for consistently caching dynamic web data," in *Proc. of IEEE INFOCOM'99*, Mar. 1999.

[7] P. Cao and S. Irani, "Cost-aware www proxy caching algorithms," in *Proc. of the USENIX Symp. on Internet Technologies and Systems*, Dec. 1997, pp. 193–206.

[8] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell, "A hierarchical internet object cache," Tech. Report 95-611, Computer Science Dept., Univ. of Southern California, Los Angeles, California, Mar. 1995.

[9] J. Dilley, M. Arlitt, and S. Perret, “Enhancement and validation of the squid cache replacement policy,” in *4th International Web Caching Workshop*, San Diego, California, Mar 1999.

[10] L. Fan, P. Cao, J. Almeida, and A. Broder, “Summary cache: A scalable wide-area web caching sharing protocol,” in *SIGCOMM*, 1998.

[11] R. Caceres, F. Douglis, A. Feldman, and M. Rabinovich, “Web proxy caching: The devil is in the details,” in *Proc. of the Workshop on Internet Server Performance*, Madison, WI, Jun 1998.

[12] H. Zhu and T. Yang, “Class-based cache management for dynamic web contents,” in *IEEE INFOCOM*, 2001.

[13] K. Rajamani and A. Cox, “A simple and effective caching scheme for dynamic content,” Tech. Report TR 00-371, Computer Sc. Dept. at Rice Univ., 2000.

[14] Squid Web Proxy Cache, “Squid-2.3,” http://www.squid-cache.org/.

[15] The Apache Software Foundation, “Apache-1.3.9,” http://www.apache.org/.

[16] JESred, “A configurable redirector for the squid internet object cache,” http://ivs.cs.uni-magdeburg.de/ elkner/webtools/jesred/.

[17] Squirm, “A redirector for squid,” http://squirm.foote.com.au/.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext transfer protocol - http/1.1,” in *Network Working Group RFC 2616*, Jun 1999.